Itérer avec confiance

Jean-Christophe Filliâtre, Mário Pereira

Itérer avec confiance

Jean-Christophe Filliâtre[1] & Mário Pereira[1] *

1: Lab. de Recherche en Informatique,
Univ. Paris-Sud, CNRS, Orsay, F-91405
et Inria Saclay – Île-de-France, Orsay, F-91893

Résumé

Un curseur est une structure de données qui permet d'obtenir successivement les éléments d'une collection, tout en laissant à l'utilisateur le contrôle de cette itération. Cet article propose une spécification formelle de ce qu'est un curseur, dans le contexte de la vérification déductive, avec le double objectif de prouver que des curseurs sont correctement réalisés et correctement utilisés. En particulier, nous proposons une approche modulaire, où le code utilisant un curseur peut être prouvé sans connaître le détail d'implémentation de ce curseur. Ce travail est validé expérimentalement en utilisant l'outil Why3, au travers de nombreux exemples.

1. Introduction

L'itération est un concept très bien connu de tous les programmeurs. C'est l'itération qui nous permet d'écrire n'importe quel algorithme, car c'est elle qui rend un langage Turing-complet. Ce concept peut apparaître sous différents aspects : des boucles, des fonctions récursives, etc. Un cas particulier d'itération se présente quand on est en train d'itérer sur une structure de données sans avoir aucun renseignement sur sa représentation interne ou même sans savoir comment le parcours est effectué. Dans ce cas-là, le processus d'itération se base sur l'utilisation des *itérateurs*.

On distingue deux classes d'itérateurs : (i) les itérateurs d'ordre supérieur ou à grand pas, qui appliquent une certaine opération à tous les éléments de la collection ; (ii) les itérateurs à petits pas qui, à chaque appel, renvoient le prochain élément (s'il existe). Dans cet article, nous nous intéressons à cette seconde catégorie d'itérateurs, encore appelés *curseurs* dans la littérature algorithmique. Plus précisément, nous nous focalisons sur des curseurs qui se présentent comme des structures impératives, tels que ceux que l'on trouve dans les bibliothèques standards C++ ou Java. Cela signifie que l'opération qui renvoie le prochain élément de l'itération, appelons-la `next` par la suite, avance à l'élément suivant par un effet de bord[1].

Notre objectif est de spécifier formellement ce qu'est un curseur, dans un contexte de vérification déductive [5]. En particulier, on souhaite être capable de prouver qu'un curseur est correctement implémenté, mais aussi être capable de prouver la correction d'un code client qui utilise un curseur. On prends en compte tous les aspects suivants.

- L'énumération ne correspond pas nécessairement au parcours d'une structure de données. Ce peut être par exemple le résultat d'un *algorithme*, tel que l'énumération de tous les nombres premiers. Dans la suite de cet article, on utilise le mot « collection » pour désigner l'information à l'origine de l'énumération, même quand il ne s'agit pas d'une structure de données.

∗. Ce travail est en partie soutenu par la Fondation des Sciences et Technologie Portugaise (bourse FCT-SFRH/BD/99432/2014) et par l'Agence National de Recherche Française (projet VOCAL ANR-15-CE25-0008).

1. Ce n'est pas une obligation. Un curseur peut être réalisé par une structure persistante [4].

- Quand il s'agit d'un curseur parcourant les éléments d'une structure de données, cette dernière peut être *mutable* et se pose alors la question de la modification concurrente de la structure et de son curseur. En Java, par exemple, ce problème est résolu en maintenant des numéros de version et en échouant dynamiquement avec une exception en cas de modification concurrente. Dans d'autres cas, le comportement est tout simplement indéfini. Dans notre cas, en revanche, on souhaite être capable de prouver *statiquement* qu'il n'y a pas de modification concurrente. Il reste néanmoins possible de modifier une structure en cours de parcours si le curseur fournit une opération spécifique (par exemple de modification du dernier élément renvoyé). Dans ce cas, on veut être capable de spécifier une telle opération.
- L'énumération réalisée par le curseur n'est pas nécessairement *finie*, comme dans l'exemple donné plus haut de l'énumération des nombres premiers. En revanche, lorsqu'elle est finie, on veut pouvoir prouver la terminaison du code qui utilise le curseur.
- Lorsqu'on prouve un programme qui utilise un curseur, on souhaite pouvoir faire *abstraction* de la façon dont est réalisé ce curseur. Ainsi, si un curseur réalise le parcours infixe d'un arbre, on souhaite spécifier qu'il s'agit effectivement d'un parcours infixe, mais sans exposer les détails de l'implémentation. Dans le cas où une structure de données est abstraite (par exemple, un ensemble dont on ne connaît pas la représentation), on souhaite de la même manière être pour autant capable de spécifier un curseur qui la parcourt.
- L'énumération n'est pas forcément *déterministe*. L'exemple le plus simple est celui d'un générateur de symboles frais. Du point de vue du client, la seule propriété importante est que la prochaine valeur renvoyée est bien différente de toutes les précédentes. Dans ce travail, nous cherchons une spécification qui n'impose pas nécessairement le déterminisme du curseur.

Dans cet article, nous proposons une façon de spécifier formellement un curseur qui répond à tous les critères ci-dessus. Nous la présentons à l'aide de l'outil de vérification déductive Why3 [1], mais l'idée est générale et pourrait être mise en œuvre facilement dans tout autre outil de vérification déductive.

Le reste de cet article est organisé de la façon suivante. La section 2 introduit succinctement l'outil Why3. La section 3 présente en détails la spécification formelle des curseurs. On illustre ensuite, dans la section 4, l'utilisation de ces curseurs au travers de nombreux exemples. En particulier, on montre la correction de curseurs mais aussi la correction de programmes les utilisant. Finalement, on discute de quelques travaux connexes et l'article s'achève sur quelques perspectives. Le développement Why3 présenté dans cet article peut être trouvé à l'adresse suivante : `http://www.lri.fr/~parreira/cursors.html`.

2. Le système Why3 en quelques mots

Le système Why3, ou simplement Why3, propose un ensemble d'outils qui permettent à l'utilisateur d'écrire, de spécifier formellement et de prouver des programmes. L'esprit de Why3 est d'effectuer cette vérification de la façon la plus automatique possible, en utilisant plusieurs démonstrateurs de théorèmes externes. Néanmoins, Why3 est capable d'interagir aussi avec des assistants de preuve interactifs, tels que Coq, Isabelle ou PVS, lorsqu'une obligation de preuve particulière ne peut pas être déchargée automatiquement.

Why3 propose un langage de programmation appelé WhyML [8], qui est un dialecte de ML avec des restrictions pour rendre la preuve automatique plus facile. Ce langage contient un certain nombre de caractéristiques communément trouvées dans les langages fonctionnels, telles que le filtrage, les types algébriques et le polymorphisme, mais aussi des constructions impératives, telles que des enregistrements avec champs mutables et des exceptions. Les programmes écrits en WhyML peuvent être annotés par des contrats, c'est-à-dire des pré- et postconditions. Le code lui-même peut être annoté, par exemple pour donner des invariants de boucle ou pour justifier la terminaison de boucles ou de fonctions récursives. Il est aussi possible d'ajouter des assertions intermédiaires dans le code, notamment pour faciliter la preuve automatique. Le système utilise toutes ces annotations pour générer des obligations de preuve grâce à un calcul de plus faibles préconditions.

La logique utilisée pour écrire les spécifications formelles est une extension de la logique

```
type seq 'a
function length (seq 'a) : int
axiom length_nonnegative: forall s: seq 'a. 0 ≤ length s
constant empty: seq 'a
axiom empty_length: length empty = 0
function ([]) (seq 'a) int : 'a
function snoc (seq 'a) 'a : seq 'a
axiom snoc_length: forall s: seq 'a, x: 'a. length (snoc s x) = 1 + length s
axiom snoc_get:
  forall s: seq 'a, x: 'a, i: int. 0 ≤ i ≤ length s →
  (snoc s x)[i] = if i < length s then s[i] else x
```

FIGURE 1 – Une théorie des séquences (extrait).

du premier ordre avec des types polymorphes (de rang 1), des types algébriques, des prédicats (co-)inductifs et des constructions récursives [6], ainsi qu'une forme limitée de logique d'ordre supérieur [2]. Cette logique est notamment utilisée pour construire des théories dans le but de modéliser le comportement des programmes. Ces théories sont le plus souvent axiomatiques. La figure 1 présente un fragment de la théorie des séquences fournie dans la bibliothèque standard de Why3. On y trouve le type polymorphe des séquences finies (`seq 'a`), une constante pour la séquence vide (`empty`), des symboles de fonctions (`length` pour la longueur d'une séquence, `[]` pour l'accès au i-ième élément et `snoc` pour ajouter un élément à la fin d'une séquence), ainsi que des axiomes définissant ces symboles. La bibliothèque standard de Why3 est constituée de nombreuses théories logiques de ce genre, notamment pour l'arithmétique entière et flottante, les ensembles et les dictionnaires.

Le langage WhyML permet aussi d'écrire du code fantôme [7], à savoir du code qui est utilisé à des fins de spécification et de preuve uniquement et qui peut être supprimé sans modification observable de l'exécution du programme. On peut voir le code fantôme comme habitant entre le monde logique et celui de la programmation.

L'intégralité de la bibliothèque standard, de nombreux exemples de programmes vérifiés, ainsi qu'une présentation plus détaillée de Why3 et WhyML sont disponibles sur le site du projet, `http://why3.lri.fr`. Néanmoins, il est possible de lire le reste de cet article sans en savoir plus sur Why3 pour l'instant.

3. Une spécification formelle

On suppose donnés deux types : un type `elt` pour les éléments parcourus par le curseur et un type `collection` pour la collection dont sont issus les éléments.

```
type elt
type collection
```

Le mot « collection » est à prendre ici dans un sens large. Il ne désigne pas nécessairement une structure de données, mais plutôt l'ensemble des informations nécessaires à la spécification du parcours. Le curseur est modélisé par le type suivant :

```
type cursor = {
  ghost mutable    visited: seq elt;
                collection: collection; }
```

Le champ fantôme `visited` contient la séquence des éléments qui ont déjà été énumérés par le curseur. Le second champ, `collection`, contient la collection dont est issue le curseur. Notre formalisation consiste à introduire ensuite deux prédicats `coherent` et `has_next` :

```
predicate coherent cursor
predicate has_next cursor
```

Le prédicat `coherent` est le cœur de notre formalisation. Il spécifie la relation entre les éléments de `visited` et les éléments de la collection. Typiquement, lorsque la collection s'apparente à un ensemble, le prédicat `coherent` indique que les éléments de `visited` font partie de cet ensemble. Le prédicat `coherent` peut aussi apporter plus d'information dans certains cas. Si par exemple les éléments sont totalement ordonnés et qu'il est spécifié que le curseur donne les éléments dans l'ordre croissant, alors le prédicat `coherent` spécifie que `visited` est trié par ordre croissant. La définition de `coherent` dépend de la nature du curseur. Prenons l'exemple d'un curseur pour parcourir les éléments d'un tableau (auquel cas le type `collection` est `array elt`). Alors, le prédicat `coherent` exprime que `visited` est un préfixe du tableau [2] :

```
predicate coherent (c: cursor) =
  length c.visited ≤ length c.collection ∧
  forall i. 0 ≤ i < length c.visited → c.visited[i] = c.collection[i]
```

Le second prédicat, `has_next`, exprime le fait que l'énumération n'est pas terminée. Là encore, sa définition dépend de la nature du curseur. Si on reprend l'exemple du tableau, alors `has_next` exprime que `visited` contient moins d'éléments que le tableau :

```
predicate has_next (c: cursor) =
  length c.visited < length c.collection
```

Les prédicats `coherent` et `has_next` étant donnés, on peut maintenant introduire les deux opérations du curseur. La première, `has_next`, a le contrat suivant :

```
val has_next (c: cursor) : bool
  requires { coherent c }
  ensures  { result ↔ has_next c }
```

Dit autrement, elle décide si le prédicat `has_next` est vrai. On note qu'elle exige par ailleurs que le curseur soit cohérent pour pouvoir être appelée. La seconde opération, `next`, a pour sa part le contrat suivant :

```
val next (c: cursor) : elt
  requires { coherent c }
  requires { has_next c }
  writes   { c.visited }
  ensures  { coherent c }
  ensures  { c.visited = snoc (old c.visited) result }
```

Comme on le voit, on ne peut appeler `next` que s'il reste encore des éléments à énumérer et si le curseur est cohérent. La postcondition assure que le curseur est maintenu dans un état cohérent et que l'élément renvoyé a été ajouté à la fin de la séquence `visited`.

En pratique, il faut aussi fournir une opération pour construire le curseur initialement. Toujours avec l'exemple du tableau, une telle opération aura le contrat suivant :

```
val array_cursor (a: array elt) : cursor
  ensures { coherent result }
  ensures { c.visited = empty }
  ensures { c.collection = a }
```

Elle renvoie un curseur dont la séquence `visited` est vide, cohérent donc prêt à être utilisé et pour lequel la collection est le tableau lui-même.

Exemple de code client. Pour illustrer l'utilisation de ces curseurs, prenons l'exemple d'un programme qui parcourt les éléments d'un tableau pour en faire la somme. (Bien entendu, on pourrait se contenter ici d'une boucle `for` sur les indices pour un tel parcours, mais l'idée est d'illustrer ici le parcours d'une collection quelconque.) On lui donne le contrat suivant, en utilisant la fonction `sum` de la bibliothèque standard de Why3 :

2. Pour des raisons de clarté, on s'autorise à surcharger les opérations `length` et `[]` pour s'appliquer autant aux séquences qu'aux tableaux, même si Why3 ne supporte pas (encore) la surcharge.

```
let sum (a: array int) : int
  ensures  { result = sum 0 (length a) (\i. a[i]) }
```

La fonction `sum` a le type `int → int → (int → int) → int`. Il s'agit d'un exemple d'utilisation d'ordre supérieur dans la logique de Why3. L'entier `sum` a b f est égal à $\sum_{a \leq i < b} f(i)$. La syntaxe `\i. a[i]` est celle de la fonction anonyme $\lambda i.\ a[i]$.

Le programme démarre avec la création du curseur et d'une référence `s` contenant la somme. À chaque itération, la valeur de `s` est mise à jour avec la valeur renvoyée par le curseur.

```
= let c = array_cursor a in
  let s = ref 0 in
  while has_next c do
    invariant { coherent c }
    invariant { !s = sum 0 (length c.visited) (\i. a[i]) }
    variant   { length a - length c.visited }
    s += next c
  done;
  !s
```

La boucle contient deux invariants. Le premier assure que le curseur est toujours cohérent, pour qu'on puisse notamment appeler les fonctions `has_next` et `next`. Le second spécifie que la référence `s` contient la somme de tous les éléments déjà énumérés. Par ailleurs, on prouve facilement la terminaison de cette boucle en majorant avec la mesure `length a - length c.visited` le nombre d'itérations restantes. Toutes les obligations de preuve générées pour ce programme sont automatiquement prouvées.

Modification de la collection. Si la collection est mutable, rien n'empêche a priori qu'elle soit modifiée alors qu'un curseur pour la parcourir est en cours d'utilisation. Ainsi, on peut imaginer le code suivant

```
let c = array_cursor a in
a[0] ← 42;
let x = next c in
...
```

qui modifie un tableau `a` après avoir créé un curseur `c` pour ce tableau. Cependant, au moment de prouver ce programme, on ne parviendra pas à montrer la précondition de la fonction `next`, à savoir `coherent c`, car le tableau a été modifié entre temps et donc le curseur également car il contient le tableau dans son champ `collection`. Si on souhaite pouvoir modifier la collection pendant le parcours, on peut imaginer que le curseur fournit une opération spécifique pour cela, par exemple

```
val update (c: cursor) (v: elt) : unit =
  requires { coherent c }
  requires { length c.visited > 0 }
  writes   { c.visited, c.collection }
  ensures  { coherent c }
  ensures  { c.collection[length c.visited - 1] = v }
```

Cette opération modifie à la fois le curseur et le tableau. Le plus important ici est qu'elle maintienne la propriété `coherent c`, ce qui nous permet de continuer à utiliser le curseur.

Hors du contexte de la preuve de programme, une telle vérification de cohérence entre le curseur et la collection est typiquement réalisée par des tests dynamiques portant sur des numéros de version. En cas d'incohérence, une exception est levée. C'est ainsi le cas dans la bibliothèque standard de Java. Dans notre contexte, en revanche, on a garanti statiquement la cohérence entre le curseur et la collection, par un ensemble d'obligations de preuve. Il n'y a donc nul besoin de tests dynamiques, ni d'exceptions. En particulier, on a donc un code légèrement plus efficace.

Énumération finie. Il est important de constater que notre spécification d'un curseur n'impose absolument pas que l'énumération soit finie. En effet, on peut tout à fait imaginer un prédicat `has_next` qui est toujours vrai (et donc une fonction `has_next` qui renvoie toujours `true`). Un des exemples le plus simples est celui d'un curseur qui énumère tous les entiers naturels.

Si on souhaite restreindre notre spécification à des énumérations finies, on peut le faire en ajoutant par exemple une notion de majorant à la longueur de la séquence `visited` :

```
function size collection : int
axiom size_nonneg: forall t. size t ≥ 0
axiom size_length_visited: forall c.
  coherent c → length c.visited ≤ size c.collection
```

On utilise ici la terminologie `size`, toujours avec l'idée d'une collection qui représente une structure de données, mais le concept est plutôt celui de longueur maximale de l'énumération. En particulier, on peut utiliser la quantité `size c.collection - length c.visited` pour justifier la terminaison d'une boucle qui utilise un tel curseur fini.

Dans l'exemple d'un tableau donné plus haut, c'est la longueur du tableau qui a joué ce rôle. On pourrait alors définir

```
function size (a: array elt) : int = length a
```

et prouver facilement les deux axiomes `size_nonneg` et `size_length_visited`.

4. Études de cas

Dans cette section, on présente plusieurs exemples de curseurs et de codes clients associés, ainsi que leur preuve de correction. L'idée est notamment de montrer que les curseurs, tels que nous les spécifions, ne sont pas limités à des structures de données. Toutes les conditions de vérification générées pour les programmes présentés dans cette section ont pu être déchargées automatiquement en utilisant une combinaison des trois démonstrateurs SMT Alt-Ergo, Z3 et CVC4.

4.1. Générateur de symboles

Le premier exemple est celui d'un générateur de symboles, c'est-à-dire celui d'un curseur qui renvoie une séquence de valeurs toutes différentes les unes des autres. La bibliothèque de Why3 fournit un prédicat `distinct`, spécifiant que les éléments d'une séquence sont deux à deux distincts, ce qui nous permet d'écrire le prédicat `coherent` aussi facilement que

```
predicate coherent (c: cursor) = distinct c.visited
```

et le prédicat `has_next` encore plus facilement :

```
predicate has_next (c: cursor) = true
```

Bien entendu, on suppose ici que le type `elt` est infini, sans quoi il ne serait pas possible de réaliser un tel curseur.

Si on prend l'exemple d'un générateur pour les entiers naturels, une réalisation possible serait

```
type elt = int
type cursor = {
  ghost mutable    visited: seq elt;
        mutable          n: int;
                collection: unit; }
```

où le champ `n` contient le prochain entier à renvoyer. Le champ `collection` ne joue pas de rôle ici, mais on le conserve néanmoins pour rester dans le cadre de notre spécification. Il est facile de réaliser la fonction `next`, en incrémentant le compteur `n`.

```
let next (c: cursor) : elt
  ...
= let x = c.n in
  c.n ← c.n + 1;
  ghost c.visited ← snoc c.visited x;
  x
```

Pour prouver ce code, il faut renforcer le prédicat `coherent` pour exprimer le lien qui existe entre `n` et `visited`.

```
predicate coherent (c: cursor) =
  distinct c.visited ∧
  forall i. 0 ≤ i < length c.visited → c.n > c.visited[i]
```

On peut voir cette second partie de `coherent` comme un invariant de liaison dans le raffinement du type `cursor` qu'on a réalisé en y ajoutant le champ `n`.

4.2. Fusion

Le deuxième exemple est celui d'un curseur qui réalise la fusion des éléments donnés par deux autres curseurs, en supposant les éléments munis d'un ordre total (sous la forme d'un prédicat `le`). Cet exemple est donc à la fois la réalisation d'un curseur et un code client qui utilise des curseurs. Le type du curseur fusion est le suivant [3] :

```
type cursor_m = {
  ghost mutable visited_m: seq elt;
                collection: (cursor, cursor);
        mutable     next1: option elt;
        mutable     next2: option elt; }
```

Le champ `collection` contient les deux curseurs dont les éléments sont fusionnés. Le champ `next1` (resp. `next2`) contient le prochain élément du curseur `c1` (resp. `c2`) qui devra être ajouté dans la fusion, le cas échéant. Le prédicat `has_next` vaut donc `true` si et seulement si `c1.next` ou `c2.next` n'est pas `None` :

```
predicate has_next_m (c: cursor_m) = (c.next1 ≠ None) || (c.next2 ≠ None)
```

La cohérence du curseur fusion est la conjonction de plusieurs propriétés. On commence par exprimer que les deux curseurs `c1` et `c2` sont eux-mêmes cohérents :

```
predicate coherent_m (c: cursor_m) =
  let c1, c2 = c.collection in
  coherent c1 ∧ coherent c2 ∧
```

On indique que la séquence des éléments déjà énumérés est triée pour la relation `le` :

```
  sorted c.visited_m ∧
```

Les éléments `next1` et `next2` font partie des éléments déjà énumérés par `c1` et `c2`, respectivement.

```
  (forall x. c.next1 = Some x → mem x c1.visited) ∧
  (forall x. c.next2 = Some x → mem x c2.visited) ∧
```

(On pourrait être plus précis et dire qu'il s'agit en réalité du dernier élément de chaque séquence, mais c'est inutile.) Par ailleurs, les valeurs de `next1` et `next2` sont toutes deux plus grands que tous les éléments déjà énumérés :

```
  (forall x i. 0 ≤ i < length c.visited_m →
    c.next1 = Some x  → le c.visited_m[i] x) ∧
  (forall x i. 0 ≤ i < length c.visited_m →
    c.next2 = Some x → le c.visited_m[i] x) ∧
```

3. On utilise ici des noms différents, `cursor_m` et `visited_m`, pour les distinguer de ceux des deux curseurs dont on fait la fusion.

Enfin, si la valeur de `next1` (resp. `next2`) est égale à `None`, alors on sait que l'énumération de `c1` (resp. `c2`) est terminée.

```
c.next1 = None → not (has_next c1) ∧
c.next2 = None → not (has_next c2)
```

On construit le curseur fusion en premier lieu avec la fonction suivante :

```
let cursor_merge (c1 c2: cursor) : cursor_m
  requires { coherent c1 ∧ coherent c2 }
  requires { c1.visited = empty ∧ c2.visited = empty }
  ...
= { collection = (c1, c2); next1 = advance c1; next2 = advance c2;
    visited_m = empty }
```

où les champs `next1` et `next2` sont initialisés avec une fonction `advance` qui renvoie le premier élément d'un curseur, lorsqu'il existe :

```
let advance (c: cursor) : option elt =
  if has_next c then Some (next c) else None
```

(On a omis ici la spécification de `advance`, qui est très proche du code.) Pour terminer, on présente la fonction `next` :

```
let next (c: cursor_m) : elt
  ...
= let c1, c2 = c.collection in
  match c.next1, c.next2 with
  | None, None → absurd
  | None, Some x2 → advance2 c; c.visited_m ← snoc c.visited_m x2; x2
  | Some x1, None → advance1 c; c.visited_m ← snoc c.visited_m x1; x1
  | Some x1, Some x2 →
    if le x1 x2 then begin advance1 c; c.visited_m ← snoc c.visited_m x1; x1 end
    else begin advance2 c; c.visited_m ← snoc c.visited_m x2; x2 end
  end
```

La fonction commence par tester les valeurs qui sont dans `next1` et `next2`. Si l'une des deux est `None`, cela signifie que le parcours du curseur correspondant est déjà terminé. On ajoute alors à la séquence `visited_m` le premier élément de l'autre curseur, qui est mis à jour avec `advance`. Dans le cas où `next1` et `next2` sont tous les deux différents de `None`, on choisit l'élément le plus petit des deux et on met à jour le champ `next` correspondant avec `advance`. Le cas où les deux énumérations sont déjà terminées est absurde, car `next` exige `has_next` en précondition. La construction `absurd` de Why3 traduit le caractère inatteignable du point de programme correspondant, en exigeant de l'utilisateur une preuve de faux.

4.3. Arbres

Notre prochain exemple est celui d'un curseur pour effectuer le parcours infixe d'un arbre binaire. Le type polymorphe des arbres binaires, `tree 'a`, est celui de la bibliothèque standard de Why3, défini comme

```
type tree 'a = Empty | Node (tree 'a) 'a (tree 'a)
```

Pour spécifier notre curseur, on commence par définir la séquence de tous les éléments d'un arbre dans l'ordre infixe :

```
function elements (t : tree elt) : seq elt = match t with
  | Empty      → empty
  | Node l x r → elements l ++ cons x (elements r)
end
```

On peut alors définir les prédicats `coherent` et `has_next` de la façon suivante :

```
predicate coherent (c: cursor) =
  prefix c.visited (elements c.collection)

predicate has_next (c: cursor) =
  length c.visited < length (elements c.collection)
```

où `prefix` est un prédicat qui indique qu'une séquence est le préfixe d'une autre séquence.

Implémentation du curseur. Pour construire le curseur, on utilise la structure de *zipper* [9], que l'on spécialise ici car on n'effectuera uniquement que des descentes vers le sous-arbre gauche.

```
type zipper = Done | Next elt (tree elt) zipper
```

Ce type représente la branche gauche de l'arbre restant à parcourir comme une liste, de bas en haut. Chaque élément de cette liste est une paire formée d'un élément et de son sous-arbre droit. Le curseur proprement dit contient le *zipper* dans un champ mutable.

```
type cursor = {
  ghost mutable    visited: seq elt;
        mutable     zipper: zipper;
                collection: tree elt; }
```

La partie intéressante de la fonction de création du curseur est l'initialisation du *zipper* :

```
let inorder_cursor (t: tree elt) : cursor =
  { visited = empty; zipper = zipper_build t Done; collection = t }
```

où la fonction `zipper_build` construit le *zipper* en descendant le long de la branche gauche d'un arbre :

```
function zipper_build (t: tree) (e: zipper) : zipper = match t with
  | Empty      → e
  | Node l x r → zipper_build l (Next x r e)
end
```

La fonction `has_next` teste si le *zipper* est égal à `Done` pour savoir si l'énumération est terminée :

```
let has_next (c: cursor) : bool
  ...
= c.zipper ≠ Done
```

Pour la fonction `next`, à chaque fois qu'on renvoie un nouvel élément, on doit reconstruire le *zipper* à partir du sous-arbre droit de l'élément renvoyé et du reste du *zipper* :

```
let next (c: cursor) : elt
  ...
= match c.zipper with
  | Done       → absurd
  | Next x r e → c.zipper  ← zipper_build r e;
                 ghost c.visited ← snoc c.visited x;
                 x
  end
```

Pour montrer la correction des opérations du curseur, il faut renforcer le prédicat `coherent` pour exprimer l'invariant de liaison entre les champs `visited` et `zipper`. On le fait en écrivant

```
predicate coherent (c: cursor) =
  elements c.collection = c.visited ++ zipper_elements c.zipper
```

où la fonction `(++)` est la concaténation de deux séquences et `zipper_elements` la séquence des éléments d'un *zipper*, définie par

```
function zipper_elements (e: zipper) : seq elt =
  match e with
  | Done       → empty
  | Next x r e → cons x (elements r ++ zipper_elements e)
end
```

Utilisation du curseur. Comme exemple d'utilisation de ce curseur, on choisit le problème classique consistant à déterminer si deux arbres binaires ont les mêmes éléments dans l'ordre infixe (problème connu en anglais sous le nom de *same fringe*). Autrement dit, on cherche à écrire une fonction avec le contrat suivant :

```
let same_fringe (t1 t2: tree elt) : bool
  ensures { result ↔ elements t1 = elements t2 }
```

L'idée est simple : on construit un curseur pour chacun des deux arbres et on énumère les éléments de chaque arbre tant qu'ils sont égaux. Ainsi, on interrompt le parcours dès que deux éléments diffèrent, ce qui est une solution élégante. La fonction `eq_cursors` suivante réalise cette comparaison :

```
let rec eq_cursors (c1 c2: cursor) : bool
  requires { c1.visited = c2.visited }
  requires { coherent c1 ∧ coherent c2 }
  ensures  { result ↔ elements c1.collection = elements c2.collection }
  variant  { length (elements c1.collection) - length c1.visited }
= match has_next c1, has_next c2 with
  | False, False → True
  | True, True   → next c1 = next c2 && eq_cursors c1 c2
  | _            → False
  end
```

Dans cette fonction, on demande à la fois un élément à chaque curseur et on les compare. Dans le cas où les éléments sont différents, on renvoie `false` immédiatement. Sinon, on continue à comparer les deux curseurs jusqu'au moment où, soit les deux curseurs s'épuisent en même temps (auquel cas on renvoie `true`), soit l'un des deux s'épuise avant l'autre (auquel cas on renvoie `false`). On en déduit alors facilement la fonction `same_fringe` voulue :

```
let same_fringe (t1 t2: tree) : bool =
  eq_cursors (inorder_cursor t1) (inorder_cursor t2)
```

4.4. Parcours en profondeur

Le dernier exemple qu'on présente est celui d'un curseur réalisant le parcours en profondeur d'un graphe (*DFS*) à partir d'un sommet donné, appelé *source* par la suite. On représente ici un graphe très simplement par le type `vertex` de ses sommets et la fonction `succ` qui associe à chaque sommet l'ensemble de ses voisins.

```
type vertex
function succ vertex : set vertex
```

La notion de chemin joue un rôle crucial dans la spécification de ce curseur. On la définit de la manière suivante :

```
predicate paths (v1: vertex) (s: seq vertex) (v2: vertex) =
  if length s = 0 then v1 = v2
  else mem s[0] (succ v1) ∧ s[length s - 1] = v2 ∧
    forall i. 0 ≤ i < length s-1 → mem s[i+1] (succ s[i])
```

Le prédicat `paths v1 s v2` signifie qu'il y a un chemin entre les sommets `v1` et `v2`, les sommets intermédiaires étant donnés par la séquence `s` (à l'exclusion du premier sommet `v1`). La définition

de `paths` est donnée intentionnellement en termes de quantification universelle et d'arithmétique, plutôt qu'en termes de prédicat inductif ou récursif, car cela conduit à une preuve automatique plus facile. On en déduit un prédicat d'atteignabilité qui exprime l'existence d'un chemin dans le graphe :

```
predicate reachable (v1 v2: vertex) = exists s. paths v1 s v2
```

Le parcours en profondeur réalisé par le curseur est écrit de façon tout à fait traditionnelle : on maintient une pile de sommets, ainsi qu'un ensemble de sommets marqués comme déjà atteints par le parcours. Le type du curseur matérialise cette pile et cet ensemble sous la forme de deux champs `stack` et `marked` :

```
type cursor = {
  ghost mutable visited: seq elt;
  ghost          source: elt;
        mutable   stack: list elt;
        mutable  marked: set elt;
             collection: unit;    }
```

Le champ `source` est utilisé pour conserver le sommet de départ du parcours. On peut maintenant définir le prédicat `coherent`, qui exprime les invariants du parcours en profondeur, à savoir que tous les éléments de la pile sont deux à deux distincts,

```
predicate coherent (c: cursor) =
  distinct c.stack ∧
```

que la source est toujours marquée,

```
  mem c.source c.marked ∧
```

que la source est seulement dans la pile quand le parcours démarre (et qu'il n'y a alors rien d'autre dans la pile),

```
  (mem c.source c.stack → c.stack = Cons c.source Nil) ∧
```

que la pile est disjointe des éléments déjà énumérés,

```
  inter (elements c.stack) (to_set c.visited) = empty ∧
```

que l'ensemble des éléments marqués est exactement la réunion des éléments déjà énumérés et de ceux de la pile,

```
  c.marked = union (elements c.stack) (to_set c.visited) ∧
```

que tous les éléments qui sont marqués sont atteignables depuis la source,

```
  forall v. mem v c.marked → reachable c.source v
```

et enfin que, si un sommet à déjà été énuméré, tous ses voisins sont déjà marqués

```
  forall v. mem v c.visited → forall w. mem w (succ v) →
    mem w c.marked
```

Le parcours est terminé lorsque tous les éléments atteignables depuis la source ont été énumérés. La définition du prédicat `has_next` exprime cette idée :

```
predicate has_next (c: cursor) =
  exists v. reachable c.source v ∧ not (mem v c.visited)
```

En termes de code exécutable, le parcours est terminé au moment où la pile est vide. La fonction `has_next` est donc ainsi définie :

```
let has_next (c: cursor) : bool
  requires { coherent c }
  ensures  { result ↔ has_next c }
= c.stack ≠ Nil
```

La postcondition de cette fonction énonce, à la fois, la correction de la terminaison du parcours (il s'arrête quand la pile est vide) et la complétude (il s'arrête seulement si on a déjà parcouru touts les éléments atteignables à partir de la source). La preuve de la complétude du parcours est difficile. Pour aider les démonstrateurs, on va introduire le lemme suivant

```
lemma path_stack: forall c v s.
  coherent c → paths c.source s v → not (mem v c.visited) →
  exists w. mem w c.stack ∧ reachable w v
```

qui exprime que tout sommet atteignable non encore énuméré est accessible depuis un sommet de la pile. Ce résultat se prouve avec un raisonnement par cas :

1. si le sommet `v` est dans la pile, c'est immédiat ;
2. si le sommet `v` n'est pas dans la pile mais la source est dans la pile, alors la source elle-même est le sommet `w` recherché ;
3. enfin, si ni le sommet `v` ni la source ne sont dans la pile, on a un chemin depuis l'intérieur de `visited` (la source) vers l'extérieur de `visited` (le sommet `v`) qui emprunte donc forcément une arête `v'→w` telle que `v'` est dans `visited` et `w` ne l'est pas. Alors, par la propriété de `coherent`, on sait que `w` est nécessairement dans la pile.

Pour aider les démonstrateurs automatiques à effectuer ce raisonnement par cas, on écrit une *lemma function*, c'est-à-dire un programme fantôme qui termine et n'a pas d'effet de bord observable, dont le contrat sera automatiquement traduit en l'énoncé donné plus haut :

```
let lemma path_stack (c: cursor) (v: elt) (s: seq elt)
  requires { coherent c }
  requires { paths c.source s v }
  requires { not (mem v c.visited) }
  ensures  { exists w. mem w c.stack ∧ reachable w v }
= if mem v c.stack then assert { paths v S.empty v } else
  if mem c.source c.stack then () else
  let _ = intermediate_value (\x. mem x c.visited) c.source v s in ()
```

La fonction `intermediate_value` est une autre fonction fantôme, utilisée pour trouver le premier sommet d'un chemin qui ne vérifie plus une propriété `p` donnée, en sachant que le premier sommet du chemin la vérifie mais pas le dernier.

```
let rec ghost intermediate_value (p: vertex → bool) (u v: vertex)
  (s: seq vertex): (vertex, vertex, seq vertex, seq vertex)
requires { p u ∧ not (p v) ∧ paths u s v }
ensures  { let (u', v', s1, s2) = result in p u' ∧ not (p v') ∧
             paths u s1 u' ∧ paths v' s2 v ∧ mem v' (succ u') }
variant  { length s }
= if length s = 0 then absurd
  else if p s[0] then
    let (u', v', s1, s2) = intermediate_value p s[0] v s[1 ..] in
    (u', v', cons s[0] s1, s2)
  else (u, s[0], empty, s[1 ..])
```

La fonction `next` est, paradoxalement, plus facile à prouver que la fonction `has_next`. On omet son code ici par manque de place, mais il est tout à fait classique.

Comme exemple de code client qui utilise le curseur DFS, on écrit un programme qui décide s'il existe un chemin entre deux sommets donnés :

```
exception Path

let is_path (v w: elt) : bool
  ensures  { result ↔ reachable v w }
  diverges
= let it = cursor_dfs v in
```

```
try
  while has_next it do
    invariant { coherent it ∧ not (mem w it.visited) }
    let x = next it in
    if x = w then raise Path
  done;
  False
with Path → True end
```

L'invariant de boucle assure que le curseur reste cohérent et qu'on n'a pas encore atteint le sommet `w`. Si la boucle termine, l'invariant assure donc que `w` n'est pas un sommet atteignable depuis `v`. Il est important de noter ici que le graphe peut être infini et qu'il n'est donc pas possible de prouver, en toute généralité, la terminaison de cette boucle et donc de cette fonction. Le mot-clé `diverges` de Why3 indique justement cette possibilité de non terminaison.

5. Travaux connexes

L'idée de spécifier et de prouver formellement des curseurs n'est pas nouvelle. Weide présente une spécification formelle pour le comportement des curseurs [14] dans le langage RESOLVE [10]. Dans ce travail, une collection est modélisée par un ensemble fini (au sens mathématique) et un curseur est spécifié par une séquence `past` correspondant à notre `visited` et une autre séquence `future` correspondant aux éléments restant à énumérer. Une troisième séquence, `original`, contient l'ensemble de tous les éléments de la collection. Avec cette conception, un curseur ne peut être utilisé que pour des collections finies et le parcours est nécessairement déterministe. L'auteur présente aussi un mécanisme pour assurer la cohérence, à savoir des opérations supplémentaires sur les curseurs, `Start_Iterator` et `Finish_Iterator`, qui doivent encadrer tout utilisation du curseur. À la différence de notre approche, la validité du curseur n'est vérifiée qu'une fois le parcours complètement terminé.

Leino et Monahan [11] présentent la spécification et preuve formelle de la mise en œuvre d'un curseur et d'un programme client. Le curseur est modélisé par un indice, qui est utilisé ensuite pour accéder au i-ième élément de la collection. Une collection est modélisée comme une séquence finie d'éléments. Une telle formalisation n'est pas vraiment abstraite, parce qu'elle ne permet que des curseurs pour des collections finies avec une notion d'accès direct au i-ième élément et uniquement des parcours déterministes.

Une méthodologie pour vérifier des instances de l'idiome *observer pattern* est proposée par Leino et Schulte [12]. Cet idiome consiste en un objet *subject* contenant des données (qui vont potentiellement changer au cours du temps) et des objets *observer* qui se servent de l'information qui est dans le *subject*. Les curseurs sont un exemple d'application de ce schéma. La contribution de ce travail est l'utilisation des *history invariants*, un style de raisonnement proche du *Rely/Guarantee*, pour donner un invariant au *subject* spécifiant son évolution au cours de l'exécution. Pour prouver la cohérence entre les éléments d'une collection (le *subject*) et un curseur associé (l'*observer*), chaque collection possède un numéro de version et un *history invariant* est utilisé pour spécifier que, si ce numéro de version n'est jamais changé, alors les éléments de la collection à ce moment-là sont les mêmes que ceux de la collection initiale. La vérification de la cohérence est faite statiquement en exigeant, dans la précondition de `next`, que les numéros de versions coïncident. Comme dans notre cas, la cohérence est donc assurée statiquement. Notre approche reste cependant plus simple, car elle ne nécessite pas de numéros de versions.

Dans la littérature, on trouve beaucoup de formalisation et de preuve de curseurs dans le cadre plus générale de la vérification de bibliothèques de structures des données. L'exemple le plus impressionnant est celui de la bibliothèque EiffelBase2 [13], une bibliothèque de conteneurs pour le langage Eiffel. La vérification est effectuée avec le système AutoProof. Il n'y a cependant aucune abstraction. Chaque curseur est écrit et vérifié indépendamment.

Un autre exemple de formalisation d'une bibliothèque de conteneurs est présenté par Dross et al [3]. Cette bibliothèque est écrite en Ada et la formalisation et la preuve sont effectuées avec les systèmes Why et Coq. Les curseurs sont modélisés par deux fonctions, *Left* et *Right*, qui représentent respectivement les éléments déjà énumérés par un curseur et les éléments restant à

énumérer. De cette façon, le parcours d'un conteneur est toujours déterministe et fini. Une autre différence avec notre travail est la présence d'une fonction *Previous*, utilisée pour revenir en arrière dans l'énumération. Il ne serait pas difficile de modéliser une telle opération dans notre contexte. Cependant, le caractère déterministe de la spécification Ada lui permet d'assurer que *Previous* suivi de *Next* redonne bien le même élément que la première fois, alors que nous ne pourrions pas assurer cette propriété en toute généralité (mais uniquement lorsque `coherent` garantit l'unicité du parcours).

6. Conclusions et perspectives

Dans cet article, nous avons présenté une formalisation de curseurs dans le système Why3, qui nous permet de prouver, de façon modulaire, à la fois le code qui réalise un curseur et le code client qui l'utilise. On modélise leur comportement d'un curseur par la séquence des éléments déjà énumérés et un prédicat la reliant à la collection parcourue. Notre spécification permet notamment des parcours non déterministes et infinis. Néanmoins, dans le cas où la collection est finie, on fournit un moyen de prouver la terminaison du parcours.

Pour valider ce travail, on a écrit et présenté plusieurs exemples de curseurs et de code client les utilisant. Certains de ces curseurs parcourent des structures de données et d'autres correspondent à des algorithmes. Ainsi, on a présenté un curseur qui effectue un parcours en profondeur et on l'a utilisé pour prouver un programme qui décide s'il y a un chemin entre deux sommets d'un graphe. L'idée de formaliser de tels curseurs semble nouvelle, à notre connaissance.

Perspectives. Dans la plupart des langages qui fournissent des curseurs, on trouve une syntaxe agréable pour les utiliser. C'est le cas de la boucle `for (E x: s)` de Java, qui n'est rien d'autre que du sucre syntaxique pour la création et l'utilisation d'un curseur pour la collection `s`. Nous envisageons d'étendre le langage Why3 avec une boucle de la forme `for x in s with c invariant I` pour parcourir la collection `s` avec le curseur `c`. À la différence de Java, on doit ici nommer le curseur, car la séquence `c.visited` va typiquement apparaître dans l'invariant de boucle `I` de l'utilisateur. Cette nouvelle construction sera, comme dans le cas de Java, du simple sucre syntaxique pour une boucle `while`, dans laquelle `coherent c` sera automatiquement ajouté comme un invariant de boucle (en plus de `I`). Ainsi, on pourra faire la somme des éléments d'un tableau aussi simplement que

```
let s = ref 0 in
for x in a with c do invariant { !s = sum 0 (length c.visited) (\i. a[i]) }
  s += x
done;
!s
```

Un autre aspect que nous envisageons d'explorer est le lien entre itération avec curseurs et itération avec fonctions d'ordre supérieur. Notamment, on se pose deux questions : (i) peut-on spécifier un itérateur d'ordre supérieur en montrant qu'il fait le même parcours qu'un curseur ? (ii) peut-on prouver un code utilisant un itérateur d'ordre supérieur, en se ramenant à la preuve d'un code utilisant un curseur (par transformation automatique) ? Ces deux questions nous intéressent notamment dans le cadre de Why3 car il ne possède pas de fonctions d'ordre supérieur dans son langage de programmation. Ainsi, la sémantique d'un itérateur d'ordre supérieur serait définie en terme de curseur, où toutes les fonctions sont de premier ordre.

Enfin, notre formalisation des curseurs fait partie d'un projet plus ambitieux dont l'objectif est de prouver une bibliothèque d'algorithmes de graphes. De tels algorithmes utilisent abondamment des itérations (par exemple pour parcourir tous les sommets d'un graphe ou tous les voisins d'un sommet) et c'est pourquoi nous avons commencé par l'étude des curseurs. Il nous reste à faire la preuve que notre spécification des curseurs est bien adaptée à la vérification de tels algorithmes, en particulier dans un contexte où l'on cherche les preuves les plus automatiques possibles.

Remerciements. Nous remercions François Pottier et Léon Gondelman pour leur aide pendant la préparation de cet article.

Références

[1] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015. See also `http://toccata.lri.fr/gallery/fm2012comp.en.html`.

[2] Martin Clochard, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Formalizing semantics with an automatic program verifier. In Dimitra Giannakopoulou and Daniel Kroening, editors, *6th Working Conference on Verified Software : Theories, Tools and Experiments (VSTTE)*, volume 8471 of *Lecture Notes in Computer Science*, pages 37–51, Vienna, Austria, July 2014. Springer.

[3] Claire Dross, Jean-Christophe Filliâtre, and Yannick Moy. Correct Code Containing Containers. In *5th International Conference on Tests and Proofs (TAP'11)*, volume 6706 of *Lecture Notes in Computer Science*, pages 102–118, Zurich, June 2011. Springer.

[4] Jean-Christophe Filliâtre. Backtracking iterators. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, September 2006.

[5] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5) :397–403, August 2011.

[6] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[7] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *26th International Conference on Computer Aided Verification*, volume 8859 of *Lecture Notes in Computer Science*, pages 1–16, Vienna, Austria, July 2014. Springer.

[8] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[9] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :549–554, September 1997.

[10] Gregory Kulczycki, Murali Sitaraman, Joan Krone, Joseph E. Hollingsworth, William F. Ogden, Bruce W. Weide, Paolo Bucci, Charles T. Cook, Svetlana Drachova-Strang, Blair Durkee, Heather K. Harton, Wayne D. Heym, Dustin Hoffman, Hampton Smith, Yu-Shan Sun, Aditi Tagore, Nighat Yasmin, and Diego Zaccai. A language for building verified software components. In John M. Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*, volume 7925 of *Lecture Notes in Computer Science*, pages 308–314. Springer, 2013.

[11] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Verified Software : Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2010.

[12] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2007.

[13] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. In Nikolaj Bjørner and Frank D. de Boer, editors, *FM 2015 : Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 414–434. Springer, 2015.

[14] Bruce W. Weide. Savcbs 2006 challenge : Specification of iterators. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems*, SAVCBS '06, pages 75–77, New York, NY, USA, 2006. ACM.